→ **Broadcasting** → The term broadcasting refers to how numpy treats arrays with different dimensions during arithmetic operations which leads to certain constraints, the smaller array is broadcast across the larger array so that they have compatible shapes.

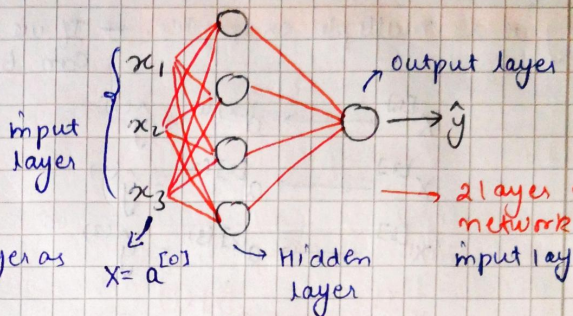
Week-3

→ Structure of a Neural Network

→ We represent the input layer as $a^{[0]} = X$

→ We represent each neuron of the hidden layer as $a_1^{[1]}, a_2^{[1]}, \ldots$

→ We represent the output neuron/layer as $a^{[2]}$

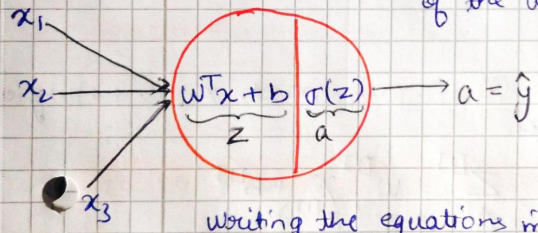


→ Therefore $a^{[1]} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ \vdots \\ a_n^{[1]} \end{bmatrix}$

- Accordingly, the weights and bias associated with hidden layers are represented by $w^{[1]}$, $b^{[1]}$.
  $w^{[1]}$ → (4×3), $b^{[1]}$ → $(4,1)$
  and with the output layer as $w^{[2]}$, $b^{[2]}$
  $w^{[2]}$ → (1×4), $b^{[2]}$ → $(1,1)$

→ Output of a Neural network

We say that, inside a neuron, two operations are performed. i.e. Calculation of z and then $\sigma(z)$. If we focus only on the first neuron of the above "2 layer NN" and following it, we get



- $z_1^{[1]} = w_1^{[1]T} x + b_1^{[1]}$ (1st node)
- $a_1^{[1]} = \sigma(z_1^{[1]})$
- $z_2^{[1]} = w_2^{[1]T} x + b_2^{[1]}$ (2nd node)
- $a_2^{[1]} = \sigma(z_2^{[1]})$
- $z_3^{[1]} = w_3^{[1]T} x + b_3^{[1]}$ (3rd node layer)
- $a_3^{[1]} = \sigma(z_3^{[1]})$

writing the equations in a matrix form we get:

$$\begin{bmatrix} — w_1^{[1]T} — \\ — w_2^{[1]T} — \\ — w_3^{[1]T} — \\ — w_4^{[1]T} — \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix} = \begin{bmatrix} w_1^{[1]T} x + b_1^{[1]} \\ w_2^{[1]T} x + b_2^{[1]} \\ w_3^{[1]T} x + b_3^{[1]} \\ w_4^{[1]T} x + b_4^{[1]} \end{bmatrix} \Rightarrow \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix} \Rightarrow Z^{[1]}$$

$w^{[1]}$ → (4×3), $b^{[1]}$ → (4×1)

Calculation of activation function $a^{[1]} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \\ \vdots \end{bmatrix} = \sigma(z_1^{[1]})$

→ Therefore generalising this we get (for 2 layer NN)

given input x:

$$z^{[1]} = w^{[1]} a^{[0]} + b^{[1]}$$
$$a^{[1]} = \sigma(z^{[1]})$$

$$z^{[2]} = w^{[2]} a^{[1]} + b^{[2]}$$
$$a^{[2]} = \sigma(z^{[2]})$$

→ **Vectorizing across multiple examples:** → If we have multiple training examples, they can be represented as:

$$x^{[1]} \longrightarrow a^{[2](1)} = \hat{y}^{(1)}$$
$$x^{[2]} \longrightarrow a^{[2](2)} = \hat{y}^{(2)}$$
$$x^{[3]} \longrightarrow a^{[2](3)} = \hat{y}^{(3)}$$
$$\vdots$$
$$x^{[i]} \longrightarrow a^{[2](i)}$$

$a^{[2](i)}$: layer 2; example i

during computations, the above equations can be formulated as:

For i = 1 to m,

$$z^{[1](i)} = w^{[1]} a^{[0]} + b^{[1]}$$
$$a^{[1](i)} = \sigma(z^{[1](i)})$$
$$z^{[2](i)} = w^{[2]} a^{[1](i)} + b^{[2]}$$
$$a^{[2](i)} = \sigma(z^{[2](i)})$$

As we recall (in the end of 1st page) that $X = (n_x \times m)$ matrix consisting of various input vectors with m training examples.

$$X = \begin{bmatrix} | & | & | & & | \\ x^{(1)} & x^{(2)} & x^{(3)} & \cdots & x^{(m)} \\ | & | & | & & | \end{bmatrix}$$

$n_x \times m$

$x^{(1)}$: 1st image; $x^{(2)}$: 2nd image; $x^{(m)}$: mth image

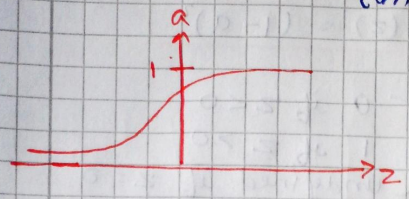
vectorizing →

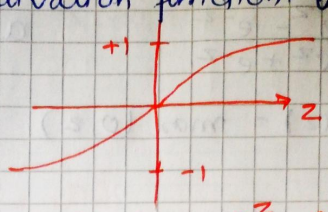
$$Z^{[1]} = w^{[1]} X + b^{[1]}$$
$$A^{[1]} = \sigma(Z^{[1]})$$
$$Z^{[2]} = w^{[2]} A^{[1]} + b^{[2]}$$
$$A^{[2]} = \sigma(Z^{[2]})$$

$$Z^{[1]} = \begin{bmatrix} | & | & | & & | \\ z^{[1](1)} & z^{[1](2)} & z^{[1](3)} & \cdots & z^{[1](m)} \\ | & | & | & & | \end{bmatrix}$$

$$A^{[1]} = \begin{bmatrix} | & | & | & & | \\ a^{[1](1)} & a^{[1](2)} & \cdots & & a^{[1](m)} \\ | & | & | & & | \end{bmatrix}$$

→ **Activation function** → as opposed to the sigmoid function, we also use "tanh" as an activation function. which works better.

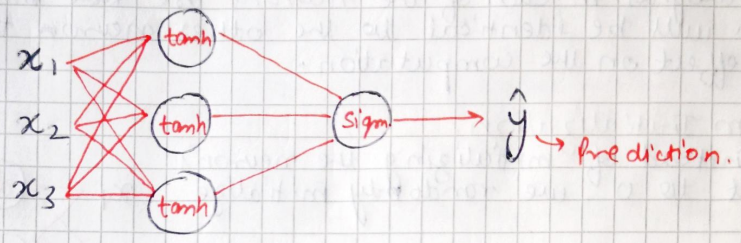




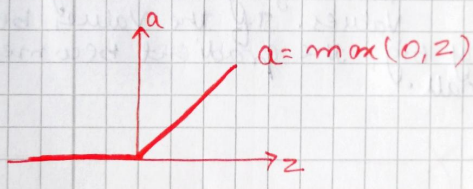
Sigmoid ⇒ $a = \frac{1}{1+e^{-z}}$

$a = \tanh = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$ { a shifted sigmoid }

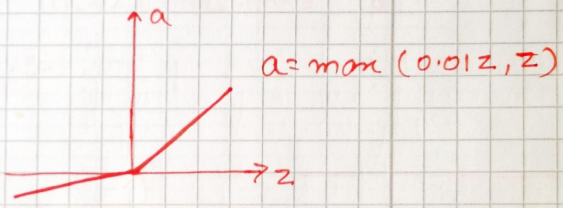
* It makes sense to use sigmoid for binary classification (because, at the output layer, we need a value b/w 0 and 1).
* i.e. activation functions can be different for different layers



→ At the higher sides (i.e. at higher values) the gradient also becomes small and therefore pushes our result to 0. therefore we also use "RELU"



RELU



Leaky RELU

→ **Importance of Non-linear activation functions**

If, instead of a non-linear activation function, we use a linear activation function, at the end it will output a linear activation function. In other words, the model is not learning anything. Instead, we are just doing a logistic progression.

* A linear hidden layer is somewhat useless.
* only, when we are using logistic regression, a linear activation function makes sense. (at the output layer)

→ **Gradient descent of activation functions**

① sigmoid $g(z) = \frac{1}{1+e^{-z}}$ $\boxed{g'(z) = g(z)(1-g(z))}$

or $a = g(z) = \frac{1}{1+e^{-z}}$ $\boxed{a'(z) = a(1-a(z))}$

② Tanh → $g(z) = \tanh(z)$ $\quad g'(z) = (1 - \tanh(z))^2$

$= \frac{e^z - e^{-z}}{e^z + e^{-z}}$ $\quad a = g(z)$ $\quad a'(z) = (1 - a)^2$

③ ReLU → $g(z) = \max(0, z)$ $\quad g'(z) \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z > 0 \\ \text{undefined} & \text{if } z = 0 \end{cases}$

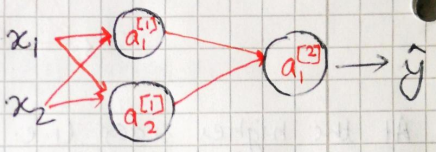
④ Leaky ReLU → $g(z) = \max(0.01z, z)$ $\quad g'(z) \begin{cases} 0.01 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$

→ **Importance of weights initialization randomly:**

If the weights in each of the neurons are ~~not~~ initialized to 0, one neuron will be identical to the other neuron thereby making no overall effect on the computation.

→ Random Initialization:

Instead of initializing the neuron's weight to 0, we randomly initialize them.



$w^{[1]} = np.random.randn((2,2)) * 0.01$

$b^{[1]} = np.zero((2,1))$

⋮ ⋮ ⋮ ⋮ ⋮ ⋮

→ initializing with very small values. If the values becomes very big, the gradient becomes very small.

Week 4 - What is a Deep Neural Network?

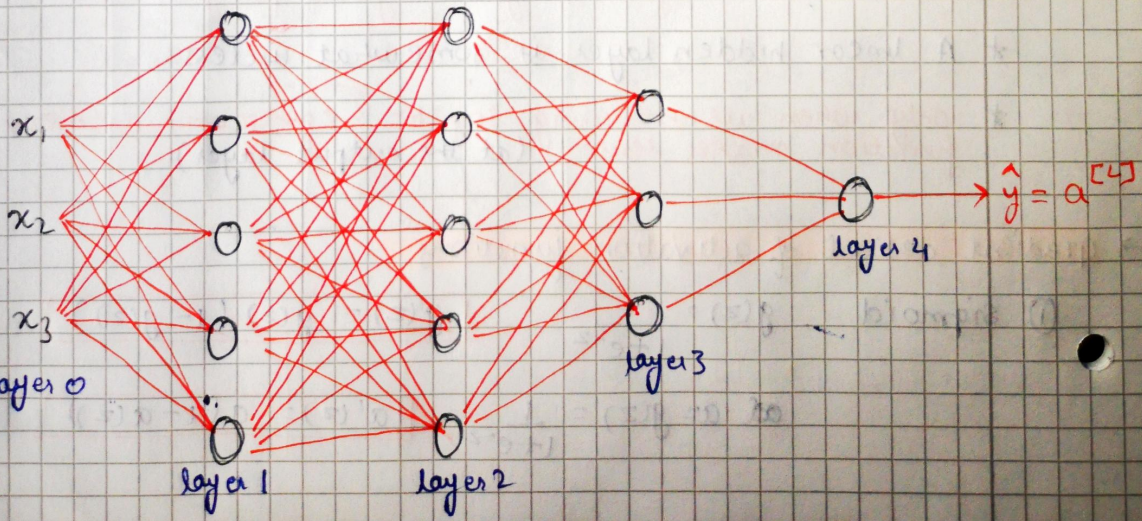
→ **Notations:**

L → no of layers

$n^{[l]}$ → no of units in each layer 'l'

$a^{[l]}$ → activations in layer 'l'

$w^{[l]}$ → weights



→ Forward prop. in DNN

for 1 training example, the calculations for the 1st layer are as follows.

$$z^{[1]} = w^{[1]} x + b^{[1]} \quad \text{------} \rightarrow / a^{[0]}$$
$$a^{[1]} = g^{[1]}(z^{[1]})$$

for 2nd layer
$$z^{[2]} = w^{[2]} a^{[1]} + b^{[2]}$$
$$a^{[2]} = g^{[2]}(z^{[2]})$$

for output layer
$$z^{[4]} = w^{[4]} a^{[3]} + b^{[4]}$$
$$a^{[4]} = g^{[4]}(z^{[4]}) = \hat{y} \text{ (Predictions)}$$

General notation:
$$z^{[l]} = w^{[l]} a^{[l-1]} + b^{[l]}$$
$$a^{[l]} = g^{[l]}(z^{[l]})$$

* in a more vectorised format for all training examples.

1st layer
$$Z^{[1]} = w^{[1]} A^{[0]} + b^{[1]}$$
$$A^{[1]} = g^{[1]}(Z^{[1]})$$

2nd layer
$$Z^{[2]} = w^{[2]} A^{[1]} + b^{[2]}$$
$$A^{[2]} = g^{[2]}(Z^{[2]})$$

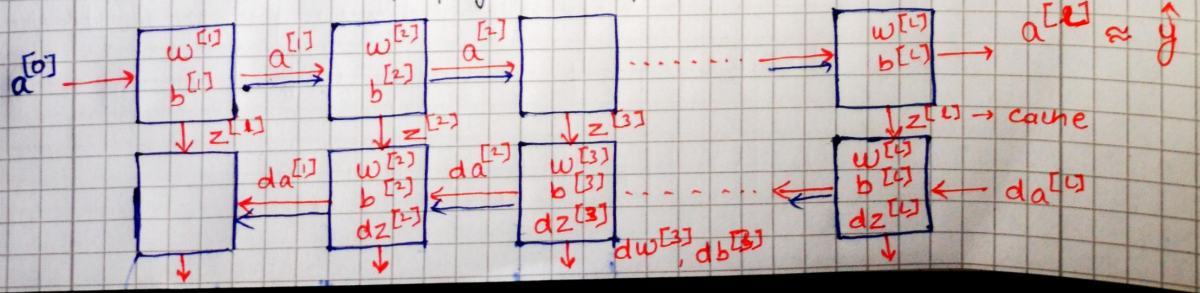
output
$$\hat{Y} = g(Z^{[4]}) = A^{[4]}$$

→ Deep Neural Networks

In a deep neural network, the earlier layers extract the low level features (eg. edges, corners) whereas, the deeper layers extract more high level features and finally they are stacked together to get the predictions. that is the reason why DNN tends to work better.

Eg. Audio → low level audio waves (initial layer) → Phenomes → words → sentences. (deep layers) → predictions.

→ Forward and Backward propogation step.



→ **Parameters and Hyperparameters**

In our model, the parameters include $w^{[1]}$, $b^{[1]}$, $w^{[2]}$, $b^{[2]}$ .....
whereas the hyperparameters include

(I) learning rate α
(II) No. of iterations
(III) .. Hidden layers (L)
(IV) Hidden units
(V) choice of activation function
⋮ ⋮ ⋮ ⋮ ⋮ ⋮ ⋮ ⋮

} these hyperparameters ultimately control the parameters.

more: momentum, mini batch size, regularizations ....
